Exhibit B E-mail from Dave Farber

[redacted] 20:18:31
dave
Exp $

From: Dave Farber <dave@sandpiper.com>
Newsgroups: sandpiper.projects.leapfrog.tech
Subject: Routing table generator
Date: [redacted] 15:54:56 -0800

This document describes a WebRepeater component called the Routing Table Generator. It includes the following sections:

- An overview of best repeater selection algorithm, including a description of the tables involved and what they do. (This is more detailed than the description of tables recently published by Louis, but only describes a few of the tables.)

- A description of the strategy we use to create and update the routing tables, identifying two separate programs. This section introduces the source of information input to these programs: the RADB (Routing Arbiter Database).

- Specific descriptions of the two algorithms that must be implemented to generate the tables. The GroupReductionTable (GRTG) is a fairly simple parser and rewriter that builds a map from IP addresses to groups; the AliasTable Generator (ATG) is a relatively sophisticated algorithm to identify clusters of connected groups.

- An Appendix gives a class interface for the Group class, which is the core of the ATG.

Overview

The "best repeater" algorithm runs on each reflector to determine how to handle each incoming request. Given the IP address of an incoming client, it must quickly determine which repeater to reflect that client to.

The best repeater algorithm does not have to be 100% accurate. In fact, as long as it not systematically worse than chance, some value will accrue to the ISP. Value to individual clients depends on how well it works for them; until the algorithm is very effective, it will work better for some clients than others. (Of course, most clients won't even no this process is happening.)

The first step is to consult a cache of recent clients. If the answer is not in the cache, the reflector makes some tests to ensure that this is a reflectable client -- requests from repeaters and "peers" are excluded and never cause reflection.

If the client turns out to be truly reflectable, the reflector then determines the "network distance" between the client and

all eligible repeaters. It combines this information with the load at each repeater, and selects the repeater with the best overall score. The way in which it combines distance and load information for a given repeater is configurable, so that an operator can adjust the trade-off between selecting a distant but unloaded repeater vs. selecting a nearby but loaded repeater.

> Note: Today the algorithm finds the repeaters with the lowest link cost and then chooses from among them the one with the lowest load. That really isn't quite right; it should be computing a "score" of link cost and load for each repeater, then choosing the repeater with the lowest score.

The units used to represent load are essentially a definition of the relative available "capacity" of each repeater; some repeaters may range between, say, 0 and 100, while other more powerful repeaters range between 0 and 1000. These numbers are in fact a combination of processing load and bandwidth load; if a system is maxed out, it doesn't matter much whether it's because it's doing too much computing or its pipes are too thin; in either case, it shouldn't be given more work.

The units for network distance (or "link cost") are discussed later in this design. Note however that this network distance has nothing to do with geographic distance; it should be a measurement that combines latency of delivery with available bandwidth. It ideally takes into account the amount of congestion between a repeater and a client. Again, there is a trade-off: a high-bandwidth but high-latency path may or may not be preferable to low-latency but low-bandwidth path.

To compute the link cost between a client IP address and a set of repeaters, the reflector performs the following steps:

1) reduce the client's IP address to a "group number," using the GroupReductionTable. This table maps ranges of IP addresses to groups.

2) reduce the group number to a "fundamental group number" using the AliasTable. This table reduces many groups to a few, representative groups.

3) using the fundamental group number, look up the cost for each repeater in the LinkStatusTable. This table maps group number and repeater number to a cost.

Tables in Best Repeater Algorithm

The GroupReductionTable, AliasTable, and LinkStatusTable are provided to the Reflector, via its contact, by the Master Repeater.

The "groups" to which IP addresses are mapped in the GroupReductionTable are Autonomous System (or "AS") numbers. AS numbers identify an independent, local adminstration

for a subnetwork within the Internet. Routers that communicate between AS's normally run a protocol called "BGP" (Border Gateway Protocol). It is a fairly safe to generalize that systems within an AS are better connected than systems from different AS's, although it is easy enough to find counter-examples. It is also generally true that the more AS's between two systems, the greater the network distance; although this doesn't have to be true, even BGP routers use this metric as an approximation. The mapping of IP addresses to AS's changes slowly over time and is the same for all ISPs. Daily or even weekly updates may be sufficient.

The AliasTable reduces group numbers to a smaller set of representative groups (called "fundamental groups"). Its purpose is to reduce a large set of groups (there are over 10,000 AS numbers) to a small and manageable set. This is important because the number of entries in link status table is equal to the number of repeaters x the number of fundamental groups. Reducing the number of entries reduces the work needed to keep the table up to date, to broadcast it to all reflectors, to store it in a reflector, and to search it in the best repeater algorithm. The reduction process is driven by knowledge of how groups are interconnected to each other and in particular to the group (or groups) in which the ISP operating the repeater sits. Therefore the algorithm to compute an AliasTable must take the ISP's location in the Internet as a parameter. The process should be run whenever the Group Reduction Table changes, and when a repeater location is added or changed.

The LinkStatusTable is generated using a heuristic that approximates the average "cost" (or "distance") from each repeater to each fundamental group. It is updated at the master by occasionally sampling a small set of real clients in a particular group, and averaging those costs.

In release one, a simple and static technique will be used to compute the cost to a given client. The cost is determined by inspecting the domain name of the client using reverse DNS, looking up the suffix in the "suffix table" to map the IP address to a "region", and looking up the region in the "region table" to retrieve an approximate score. Because of averaging, this mechanism tends to assign high scores to groups in which most clients are in a region with a high score. The clients selected randomly for measurement are chosen from the pool of real clients who have made requests to reflectors.

In release two, we will augment these techniques with measurements that use variable-length pings to determine latency and bandwidth. We can also augment these techniques with costs available from third-party tools, such as Cisco's DistributedDirector or ISI's GDNS.

A key assumption in this whole process is that averaging using average costs for a sampling of clients in a group will provide a reasonably valid estimate of the cost to all clients in the group (at least, relative to the cost to the same clients from other repeaters). Evidently, if the groups themselves are not adequately defined, these averages will also be meaningless.

Strategy

LinkStatusTable updating has been implemented for release 1, and is the task of the Master Repeater. The "Routing Table Generator" is responsible for computing the other two tables: the GroupReductionTable and the AliasTable. I will refer to these programs below as the Group Reduction Table Generator (GRT) and the Alias Table Generator (ATG).

These two programs require as input a description of the Internet. A "true" and global description of the global internet is not available anywhere, even an out-of-date description. What actually happens is that BGP (Border Gateway Protocol) routers take information from their administrators about local connectivity and routing policies, and path information from neighboring routers, and construct new paths to propagate to other routers. Because each router filters the paths that it propagates, each has only a local perspective of its environs; no global picture emerges.

Fortunately, however, Merit Network Inc. (in collaboration with USC Information Sciences Institute, and funded in part by Cisco) is providing a public service by collecting and publishing information on AS maintenance and connectivity. The RADB project provides tools uses by network administrators around the world to describe their local connectivity. The project collects databases built from these tools by several agencies world-wide, and makes them available via FTP. Among other things, the data is used by network administrators to debug routing problems and if necessary contact other network administrators. Because the information is not derived from routers, it is not necessarily correct. However, it should be a very good starting point.

Note that this information is copyrighted and may not be transmitted or stored without permission of its owners. Sandpiper Networks Inc. will need to seek permission of its owners to use the data. SNI should also build a relationship with Merit/ISI, in order to help ensure that the data remains avaiable or can be derived from another source elsewhere. See http://www.ra.net/RADB.tools.docs for more information.

RADB files

RADB files are available from ftp://ftp.ra.net/routing.arbiter/radb/dbase/*. The are updated regularly, perhaps once a day. At 9PM PST on March 6, I found

the following files in place:

```
-rw-r--r--    1 ra          457087 Mar  7 03:05 ans.db.gz
-rw-rw-r--    1 ra          141851 Mar  6 13:37 canet.db.gz
-rw-rw-r--    1 ra          802428 Mar  6 17:27 mci.db.gz
-rw-r--r--    1 ra         3090675 May  9  1995 prdb.db.Z
-rw-rw-r--    1 ra         1347789 Mar  7 03:05 radb.db.gz
-rw-rw-r--    1 ra          518672 Mar  7 03:05 ripe.db.gz
```

The Merit RADB pages describe the source of these files.

Each database is a simple text file. Records have one field per line and delimited by a blank line. Fields typically have the form "*xx: data" where "*" indicates that this is an abbreviated field name, "xx" is the abbreviated field name, and "data" is the data for the field.

There are several different record types, indicated by the first field name in the record. I believe that "an" records define AS's, and "rt" records define routes, including IP address sequences and connectivity information.

A specification available on line describes all the fields and gives their unabbreviated names, but my impression is that only abbreviated names are used in these files.

I've copied several of these files to bud://SRC/leapfrog/route for convenience. These can be used as test files to debug the routing table generator programs.

RADB questions

The RADB is a large, rich and cluttered source of data that we will want to explore. To begin with, we need a simple parser that accepts RADB files in their native form and converts all records into an internal form. We should be able to configure the record types and field types that it accepts, so that it can ignore unwanted data. We may want to output the data into a standard database (e.g. Access) so that we can experiment with it.

Some of the questions we will want to ask include the following:

- how many records typically change from day to day
- how many AS's are there now
- how many AS macros are there
- are AS's defined in several macros
- how good is the coverage mapping IP addresses to AS's
- are there inconsistencies where IP addresses map to several AS's.
- there is some connectivity information in the RADB. How much is it. Is it up to date?

We can test random elements or inconsistent elements for correctness by using traceroute to see the real scoop on a particular IP address.

Algorithms

Both the GRTG and the ATG are stand-alone programs that convert RADB files into tables used by WebRepeater. The following sections detail the design of the GRTG and ATG.

GRTG

The GRTG is a very simple program that extracts AS information from the RADB files to create a map from IP addresses to AS numbers. The GRTG looks for "rt" records and extracts from them the IP address, number of valid bits, and AS number. Here is a sample record taken from the middle of MCI's database:

```
*rt: 204.70.32.84/30
*de: MCI HAY Border 1
*or: AS3561
*wd: 941007
*mb: MCI
*mb: MCI
*ch: roy@mci.net 941007
*so: MCI
```

In this record, the "rt" field defines the IP address prefix and number of bits -- in this case the prefix has 30 bits meaning it describes a cluster of 4 adjacent addresses. The "or" field defines the origin AS number, which in this case is AS3561. The GRT has to convert this record into a one line record of the GroupReductionTable, which will have a form like this:

```
204.70.32.84 30 3561
```

There is a technique called CIDR used to compress IP addresses in routers. For instance, if another record in the RADB defines rt: 204.70.32.88/30, the two records could be combined into a record of the form 204.70.32.84/29, which defines 8 adjacent addresses. It is not clear whether the RADB contains any CIDR-compressed information, but we may want to consider it to reduce the size of the tables.

ATG

The ATG is a "clustering algorithm" that combines groups into larger clusters, until a target number of clusters is reached. It attempts to select groups to keep cluster weights approximately the same and maximize connectivity within clusters. This is based on heuristic definitions for weight and connectivity.

The clustering algorithm uses the RADB and other information to generate an AliasTable for use in selecting the best repeater.

The algorithm operates in three phases:

- Phase I: read exiting tables to initialize a set of groups
- Phase II: merge groups using several techniques
- Phase III: create and output alias maps from groups

Groups

The ATG works with objects from the class Group.
A Group instance contains the following information:

- group ID

  This can be an AS number, AS macro name, or internally assigned name. All final groups are also assigned an AS number selected from its members.

- link table

  The link table is a set of neighboring groups and the "weight" of each of them.

- set of subgroups

  The algorithm creates compound groups from subgroups. A group with no subgroups is called an "atomic" group.

- group owner

  If a group G has subgroups, G is the owner of the subgroups. A special group called the 'root' group owns all groups that are not owned by other groups.

- group weight

  The size of a group is a heuristic measure of how large it is. The weight of a group is the total number of connections it has, both internal and external. (This seems like a simple and useful guess, others might be used in the future.)

In future versions of the algorithm, we may assign a region ID or a geographic "locus" to a group to help compute the cost between groups.

An outline of the Group class is defined below. Most of the methods are straightforward. One critical and fairly complicated method, called "recompute," is described below.

Phase I: Initializing Groups

Groups are defined using information contained in RADB files, specifically, in *an records, which providing routing policy information. Each *an record defines a single AS and lists all of the other AS's to which it is connected.

A *an record has the following form:

```
*an: ASxxxx
...
*ai: ASxxxx yyy...
*ao: ASxxxx yyy...
*df: ASxxxx yyy...
```

The *an field at the beginning of the record defines both the record type and the group ID (ASxxxx, where xxxx is 1-4 digits.)

The record may contain zero or more records of type ai ("as-in"), ao ("as-out"), and df ("default"). These define linkages to other AS's. The ASxxxx component of each record defines the neighboring AS to which the current AS is attached. The parts of the line shown as "yyy" above define rules and priorities for accepting and rejecting traffic and can be ignored for our purposes. Note that the same neighbor may be listed in the ai, ao and df rules more than once.

Phase II: Merging Groups

Groups are members of the 'root group' until made members of another compound group.

Once the initial groups defined by "*an" records are created, the algorithm attempts to merge groups into larger groups, until the number of members of the root group drops to a specific target size.

Ideally the groups created have the property that, given a set of repeaters and any pair of IP addresses in the group, the same repeater would be the best choice for both of them. That goal is impossible, but a simpler goal is to just identify groups that are logically well connected within themselves and relatively poorly connected between themselves.

Several techniques are used:

1) "Macros" in RADB define groups of related AS's. Macros are defined by *am records. Here is an example record:

```
*am: AS-SESQUISTUB
*de: Single Homed Sesquinet Customer ASs
*al: AS1832 AS2712 AS302 AS3526 AS4050 AS8
*gd: hostmaster@sesqui.net
*tc: SB98
*ac: FG50
*ny: hostmaster@sesqui.net
*mb: SESQUINET-MAINT-MCI
*ch: hostmaster@sesqui.net 941229
*so: MCI
```

This step is implemented by creating a group whose ID is the macro name, and using addGroup() to merge each AS in the macro into it.

The Group class prevents any group from being a member of more than one compound group. This event will be detected if there are AS's that belong to more than one macro. The event should be logged but should not stop processing.

2) Any group which has only a single link (a "leaf") can be merged into the group to which it is linked. Occasionally merging a leaf group into another will create a new leaf group. A single pass through all of the groups can eliminate all leaves.

3) After completing the above steps, the ATG mergers smaller groups into larger ones. This is done by finding the "lightest" group, determining which of its neighbors is the best to be merged with, and merging the two. If either group is compound, the other is added to it; otherwise, a new compound group is created. The algorithm continues until a target number of total groups remains.

   The best neighbor is defined as the neighbor with the highest connectivity. For now, connectivy is a measure of the fraction of members of this group that directly connect to the neighbor.

   [Note that, given this definition, step (2) above more or less falls out of step (3).]

Phase III: Output

In phase III, an AliasTable is built. For each Group in the root Group, the ATG selects an atomic group and use its ID as the fundamental group number. It traverses all subgroups recursively, emitting a mapping entry between the id of each atomic group and the fundamental group number.

Assuming a group contains AS's 8, 302, 1832, 2719, 3527, and 4050, the AliasTable might contain the following records:

```
1832 2712
1832 302
1832 3526
1832 4050
1832 8
```

The ATG also provides a report containing information about the final groups that remain as subgroups of the root groups. The report is in the form of a table with the following columns:

group id
number of atomic members
number of cluster members

```
    total number of external links
    total number of internal links
    total weight
```

Commentary

The best neighbor algorithm can probably be improved in a number of ways. For instance, an alternative to finding the lightest group would be to look at the smallest connectivity among all root groups and merge the selected groups. This will take longer and may produce groups with more variation in weight but they should be better connected.

In addition, link weights can be replaced by a better approximation that takes into account measured distances between AS's, geographic location, or the heuristic values.

The database created can also be used to define static estimates of costs, which can be used to create a static version of the LinkStatusTable. While this approach is not strictly necessary in release one, it might be worth pursuing.

Appendix: The "Group" class
==========

Below is an outline of the Group class. Most of the routines here should be codable in just a few lines.

```
/**
 * Group represents an entity with neighbors (other groups)
 * connected by links.  The links have costs associated
 * with them, which define a distance metric between groups.
 *
 * Groups can be combined into clusters (larger groups).  There
 * is a way to get the "score" of a group, which defines how
 * well clustered its.
 */
class Group implements {

    private static Hashtable registry;
    private Group root;

    private String id;
    private Group owner;
    private Vector subgroups;

    /*
     * internalLinkWeight is total number of internal
     * links between all atomic subgroups of this group.
     */
    private int internalLinkWeight;

    /*
     * externalLinkWeight is the sum of the weights
```

```
     * of all external links, which should be the same as the
     * total number of links between all atomic subgroups
     * of this group and all groups external to this group.
     */
    private int externalLinkWeight;

    /*
     *  Links are represented using a Hashtable.
     *  A Key is a reference to a target Group,
     *  The corresponding value is a Float objects
     *  which represents the weight of the link.
     *  The weight of an external link is the
     *  number of atomic subgroups that refer to the
     *  target Group.
     *
     *  (Implementation note: since Float objects
     *  cannot be changed, and these weights are
     *  changed frequently, it might be better to make
     *  the target be an object of a new type, say "Link",
     *  which contains methods to getWeight and setWeight.)
     */
    private Hashtable links;

    /*
     * Group types
     */
    public static int TYPE_NEW = 0;
    public static int TYPE_ATOMIC = 1;
    public static int TYPE_COMPOUND = 2;
    public static int TYPE_ROOT = 3;

    /**
     * Given the id of a Group, find the Group in the registry.
     *
     * The special "root" Group has the name "root"
     */
    public static Group findGroup(String id);

    /**
     * Create an empty new group, that belongs to the root.
     * Make up an ID. Register the group in the registry.
     */
    public Group() {}

    /**
     * Create an empty new group, that belongs the root,
     * with the given ID.  Register the group in the registry.
     */
    public Group(String id) {}

    /**
     * Get the id of the group
     */
    public String getId() {}

    /**
     * Get the type of the group:
```

```
     *    TYPE_NEW if it has no links or members
     *    TYPE_ROOT if this==root
     *    TYPE_COMPOUND if it has subgroups
     *    TYPE_ATOMIC if it has links and no subgroups
     */
    public int getType() {}

    /**
     * Get the weight of this group.  The weight is the
     * sum of internalLinkWeight and externalLinkWeight.
     */
    public float getWeight() {}

    /**
     * Get the owner of this group (or null if it has no owner)
     */
    public Group getOwner() {}

    /**
     * Return true if 'subGroup' is a subgroup of this group,
     * either directly or indirectly.
     */
    public boolean contains(Group subGroup) {
        while ( subGroup != root ) {
            if (this == subGroup.owner)
                return true;
            subGroup == subGroup.owner;
        }
        return false;
    }

    /**
     * Add a link unless it already exists.
     *
     * If the link is new, set its weight to 1, and
     * increment externalLinkWeight.
     *
     * Note: should we add the corresponding link to
     * neighbor, or will separate RADB connectivity
     * information take care of it?
     *
     * Note: it may be appropriate to assign a higher
     * weight to default (*df) links.  This is not currently
     * done.
     *
     * Throws GroupException if the group is not of TYPE_ATOMIC
     */
    public void addLink(Group neighbor) throws GroupException {}

    /**
     * Get the weight of the link to neighbor.  If there is no
     * such link, the weight is zero.
     *
     * Throws GroupException if groups are not already linked.
     */
    public float getLinkWeight(Group neighbor) {}
```

```
    /**
     * Compute and return a measure of the connectivity score.
     *
     * The best neighbor algorithm that merges groups is based
     * on selecting the neighbor with maximum connectivity.
     *
     * For now, connectivy is a measure fraction of links
     * in this group that directly connect to the neighbor.
     * This measure may be replaced by others in the future.
     */
    public float getConnectivity(Group neighbor) {
        return getLinkWeight(neighbor)/externalLinkWeight;
    }

    /**
     * Get an enumeration of all links for this group.
     */
    public Enumeration getLinks() {}

    /**
     * Add a subgroup to this group.
     *
     * For each link between the subgroup and any group G,
     * if G is not a member of this group, add a link between
     * this group and G, and set the weight of that link to the
     * weight of the link between the subgroup and G.  If the
     * link already exists, increase its weight by the
     * weight of the new link.
     *
     * If G *is* a member of this group, add the weight of
     * the link to the internalLinkWeight.
     *
     * Throws GroupException if subgroup is not owned by 'root'.
     * Throws GroupException if 'this' has been assigned links
     * with addLinks().
     */
    public void addSubgroup(Group subgroup) throws
    GroupException {}

    /**
     * Remove a subgroup from this group and adjust
     * this group's weight and links to undo the action
     * of addSubgroup(). Set the owner of
     * the subgroup back to 'root'.
     *
     */
    public void removeSubgroup(Group subgroup) {}

}
```